(2)

Technical Report

CMU/SEI-89-TR-2
ESD-89-TR-2

Software Process Modeling: Principles of Entity Process Models

Watts S. Humphrey
Marc I. Kellner

February 1989

89

Technical Report
CMU/SEI-89-TR-2
ESD-89-TR-2
February 1989

Software Process Modeling: Principles of Entity Process Models

Watts S. Humphrey
Marc I. Kellner
Software Process Program

This report appears in the *Proceedings of the 11th International Conference on Software Engineering*, 15-18 May, 1989, Pittsburgh, Pennsylvania.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

[illegible]
Karl Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

List of Tables

Software Process Modeling: Principles of Entity Process Models

Abstract: A defined software process is needed to provide organizations with a consistent framework for performing their work and improving the way they do it. An overall framework for modeling simplifies the task of producing process models, permits them to be tailored to individual needs, and facilitates process evolution. This paper outlines the principles of entity process models and suggests ways in which they can help to address some of the problems with more conventional approaches to modeling software processes.

1. Introduction

Considerable attention has been devoted to software process modeling (or process programming) during the past few years. Models of software life-cycle processes are expected to provide a means of reasoning about the organizational processes used to develop and maintain software. Most efforts in this field have focused on the functional, or task-oriented, aspects of processes, although a few recent papers have proposed behaviorally oriented modeling approaches. Even these, however, still approach behavioral modeling from a task orientation, focusing on when tasks are performed. This paper proposes an entity orientation to behavioral modeling. The paper argues that entity process models can be helpful in circumventing some of the problems with more task-oriented models of software processes.

Following a discussion of the problem domain, the principles of entity process models are presented. These principles are illustrated by a detailed example of this approach, applied to a significant component of a software process. The example focuses on the coding and unit testing of a module, including many realistic feedback paths that make software processes so complex. Finally, the paper derives sample schedules from the model, both with and without resource constraints. The paper concludes with comments on the managerial value of entity process models.

2. Background

2.1. The Need for a Defined Software Process

When several people work cooperatively on a common project, they need some way to coordinate their work. For relatively small or simple tasks, this can often be done informally, but with larger numbers of people or more sophisticated activities, more formal arrangements are needed. For example, process definition can be compared with football training. Teams without defined and practiced plays do not make the play-offs. While the sequence of plays will change from game to game, the winning team generally has worked out their plays in advance, knows when to use them, and can perform them with skill. The software process is much the same. Unfortunately, few software teams work out their plays in advance, even though they know the key problems they will encounter. For some reason they act as if late requirements changes, regressions, or system integration problems will not recur.

The software process is the technical and management framework established for applying tools, methods, and people to the software task. A defined process not only prepares for likely eventualities; it also provides a mechanism for organized learning. As projects improve their methods for handling key tasks, these can be incorporated in the repertoires of "plays" available to the rest of the organization. This process definition makes it easier for each new project to build on the experiences of its predecessors, and it also protects against the dangers of ill-prepared crisis reactions. An important observation of process management is that process changes adopted in a crisis are generally misguided. Crises are the times when shortcuts are most likely and when organizations are most prone to omit critical tasks. These shortcuts often lead to truncated testing, skipped inspections, and deferred documentation. With time at a premium, rationalization is most likely and process judgments are least reliable. Because it is difficult to justify many tests, analyses, or inspections in the heat of the moment, a thoughtfully defined and approved process can be a great help. When the crisis occurs, it has been anticipated, and established means are at hand to deal with it. In short, a defined process provides the software professionals with the framework they need to do a consistently professional job.

2.2. The Need for a Standard Process Framework

While there are often needs for project-specific process tailorings, there are also compelling reasons for a standard process framework:

1. Process standardization is required to permit training, management, review, and tool support.
2. With standard methods, each project's experiences can contribute to overall process improvement in the organization.
3. Process standards provide a structured basis for measurement.
4. Because process definitions take time and effort to produce, it is impractical to produce new ones for each project.
5. Because the basic tasks are common to most software projects, a standard process framework will need only modest customization to meet most special project needs.

2.3. Software Process Models

A software process architecture is a framework within which project-specific software processes are defined [Humphrey 88]. It establishes the structure, standards, and relationships of the various process elements. Within such an architectural framework it is possible to define many specific processes. A software process model is then one specific embodiment of such a software process architecture.

While software process models may be constructed at any appropriate level of abstraction, the process architecture must provide the elements, standards, and structural framework for refinement to any desired level of detail.

2.4. Criteria for Effective Process Models

Before establishing criteria for evaluating modeling approaches, it is first necessary to define the uses for such models. The basic uses for process models are to:

1. Enable effective communication regarding the process. This could involve communication among process users, process developers, managers, or researchers. It

enhances understanding, provides a precise basis for process execution and automation, and facilitates personnel mobility.

2. Facilitate process reuse. Process development is time-consuming and expensive, so few project teams can afford the time or resources to totally develop their own.
3. Support process evolution. Precise, easily understood, expandable, and reusable process definitions provide an effective means for process learning. Good process models are thus an important element of software process improvement.
4. Facilitate process management. Effective management requires a clear understanding of plans and the ability to precisely characterize status against them. Process models potentially provide a framework for precisely defining process status criteria and measures [Kellner 88a, Kellner 88b, Kellner 89].

From this, it is clear that process models must:

A. Represent the way the work is actually (or is to be) performed.
B. Provide a flexible and easily understandable, yet powerful, framework for representing and enhancing the process.
C. Be refinable to whatever level of detail is needed.

The first two points are relatively obvious, but the third is often overlooked. As improvements are made in supporting the software process, precise task definitions are required to permit effective tool and environment development [Feiler 86, Feiler 88]. As with any data processing application, poorly understood tasks lead to inadequate requirements and ineffective systems. While it is more challenging than most application developments, software process automation is much like application development and thus must start from precise process models at relatively deep levels of detail.

Finally, to be most effective in supporting the four objectives of process modeling presented above, process models must go beyond representation. They must support comprehensive analysis of the process through the model, and allow predictions to be made regarding the consequences of potential changes and improvements. Certainly, modeling approaches that smoothly integrate representation, analysis, and predictions are preferred. We describe such an approach elsewhere [Kellner 88a, Kellner 88b, Kellner 89].

3. Process Modeling Concerns

3.1. The Problems with Current Software Process Models

A number of process models have been proposed in the literature, including the "waterfall model" [Royce 70, Royce 87], the "spiral model" [Boehm 86], and "iterative enhancement" [Basili 75]. Nevertheless, outside the research community, much software process thinking is still based on the waterfall framework, as described by Win Royce in 1970 [Royce 70]. While this model has been helpful in explaining the software development process, it has several shortcomings with respect to the above criteria [Boehm 86]:

1. It does not adequately address the pervasiveness of changes in software development (A).
2. It unrealistically implies a relatively uniform and orderly sequence of development activities (A).

3. It does not easily accommodate such recent developments as rapid prototyping or advanced languages (B).
4. It provides insufficient detail to support process optimization (C).

Over-reliance on the waterfall model has had several unfortunate consequences. First, by describing the process as the sequence of requirements, design, implementation, and test, each step is viewed as completed before the next one starts. The reality is that requirements live throughout development and must be constantly updated. Design, code, and test undergo a similar evolution. The problem is that when managers believe this unreal process, they require that design, for example, be completed before implementation starts. Everybody who has ever developed much software knows that there are many tradeoffs between design and implementation. When the design is not impacted by implementation, it means that either the design went too far or the process was too rigid to recognize and adjust for implementation problems. The design and its documentation must thus evolve in concert with implementation.

Unrealistic software process models also bias the planning and management system. When requirements are supposed to be final before design starts, various documents and reviews are conducted to demonstrate requirements completion. Since these documents must also change as design issues are exposed, this static view of requirements can be counterproductive. This is further exacerbated by pressure for an early freeze on changes. This inhibits creative design/requirements tradeoffs just when they should be encouraged. These problems have corresponding analogues in design, implementation, and test.

A final consequence is process measurement. When an unrealistic process model is used as a basis for planning, the measurement and tracking system is corrupted. Tasks are labeled as complete when they are not, so crisp progress benchmarks are not available. Since resource or lead time standards are corrupted by the lack of clear activity boundaries, planning and tracking are equally imprecise.

3.2. The Causes of Current Model Problems

The fundamental problem with current software process models is that they do not accurately represent the behavioral (or timing) aspects of what is really done (criterion A). The reason is that traditional process models are extremely sensitive to task sequence; consequently, simple adjustments can require a complete restructuring of the model. For example, consider the actual interaction between requirements, design, and implementation in many software projects. Often, an implementor will be faced with a decision on the specific presentation of some error or other message. Rather than making an arbitrary decision, such issues should be referred to a systems design group for resolution. Because many such choices involve user interaction, they actually represent requirements refinements. Clearly, the requirements that were supposed to have been completed were not.

This problem results from an overemphasis on the modeling of tasks. While this seems like a natural way to guide task-oriented people, it limits human flexibility and tends to arbitrarily impose rigidity. With the requirements-design-code-test sequence, decisions to re-examine the requirements during test, for example, cannot be readily accommodated. In short, traditional modeling has not adequately addressed the behavioral aspects of processes.

3.3. Process Modeling Considerations

First, it is important to recognize that a complete model of a complex process must be complex. Here, the operative word is complete. If one wants to use a model for a specific purpose, it can presumably be tailored to that purpose and compromise completeness in other respects. These compromises, however, must be made with care or the resulting simple model representation could easily be misleading.

The traditional task-oriented approach to process models results naturally from our task-oriented view of our work. This, for example, is what led to the waterfall model: the need for a general prescription of human activity. While this task structure is quite appropriate and relatively easy to understand when the tasks are simply connected, it becomes progressively less helpful as the number of possible task sequences increases. While it can, in principle, still produce an accurate model, it becomes more difficult to do so and progressively less understandable.

The real danger of attempting to use task-oriented models in such complex situations is that they must be simplified to permit human comprehension, and these simplifications tend to limit flexibility in task sequencing. When there are, for example, ten possible actions that could be usefully performed, a simplified task-oriented process model would presumably only show one or two. While this might be perfectly acceptable under normal circumstances, the other alternatives might then be viewed as abnormal or unauthorized. When such models are used to guide process automation, project management, or contract administration, the resulting process rigidity can cause serious problems.

The question, therefore, is not "What is the right way to model the process?" but "What is the most appropriate way to model this process for this purpose?"

A final point concerns actual process execution. When the process model is used to guide or control the detailed performance of the process by people or machines, a comprehensive model is required and it must include a task-oriented view. Indeed, we have suggested that a complete process model needs to contain functional, behavioral, structural, and conceptual data modeling views [Kellner 88b, Kellner 89]. For process management purposes, however, a simpler process model is appropriate as long as it does not artificially constrain process execution. There is good reason to believe that task-oriented models are not the best for this purpose.

Considerable attention has been devoted to software process modeling (or process programming) during the past few years; for example, see [Fourth 88, Lehman 87, Osterweil 87, Rombach 89] However, most efforts have focused on the functional, or task-oriented, aspects of processes. Although some behaviorally oriented modeling approaches have been recently reported [Phillips 88, Phillips 89, Williams 88a, Williams 88b], these efforts still approach behavioral modeling from a task orientation, focusing on when tasks are performed. This paper proposes an entity orientation to behavioral modeling, as described below.

4. Entity-Based Models

One alternative is to consider basing process models on entities, similar to those used by Jackson in the Jackson System Development (JSD) methodology [Jackson 83]. Here, one deals with real entities and the actions performed on them. Each entity is a real object that exists and has an extended lifetime. That is, entities are things that persist rather than ephemeral objects that are transiently introduced within the process.

Examples of such entities are the requirements, the finished program, the program documentation, or the design. While these sound like some of the items discussed in the waterfall model, they are quite different. The traditional waterfall model deals with tasks such as producing the requirements. This task is then presumed completed before the next one (design) starts. In reality, the requirements entity must survive throughout the process. While it undergoes many transformations, there is a real requirements entity that should be available at all later times in the process. Exactly the same is true of the design, the implementation, and the test suite.

4.1. The Stability of Entity Process Models (EPMs)

The reasons that EPMs provide a useful representation of a software process are:

- EPMs deal with real objects (entities) that persist.
- Each entity is considered by itself and is viewed as having a defined sequence of states.
- State transitions result from well defined causes, although they may depend on the states of other entities as well as process events and conditions.
- As long as the relative sequential relationships of these transitions are retained within each entity stream and as long as any prerequisites and dependencies between entities are maintained, the timing within the various entity streams is not material.

Each entity stream is thus individually robust and can be stretched or compressed without destroying its self-consistency. Further, as one entity stream is adjusted, one need only consider those other process aspects that are involved in the transition conditions between entity states.

4.2. Entities

The proper definition of the entities in a process is important. An entity must:

- Exist in the real world and not merely within the model or process.
- Be identifiable and uniquely named.
- Be transformed by the process through a defined set of states [Jackson 83].

Entities are real things, not just artifacts that are introduced to assist in the work. When, for example, the full software product life cycle is considered, a number of the "artifacts" produced during development become extremely important. Examples are the requirements, the design, the test documents, the test cases, and the test plans. When products evolve through multiple versions, similar needs arise. What is not so clearly recognized, however, is that the traditional view of these items as end products of a development phase can lead to the belief that they are completed and will undergo no further changes. As a result, there are often no official process provisions for keeping the design up-to-date throughout implementation and test.

4.3. Software Process Entities

This specification of an entity requires that we define the "real world." For purposes of software development, we define the real world as the set of all users for the software system. This does not include, for example, software development itself nor the management and contracting activities that drive and control it. This is arguably subjective, but the logic is to separate those essential products of the process from the artifacts and mechanisms used to control and produce it.

The question then is, what are the potential uses for the software product and what process artifacts are required to support them? Some obvious entities are:

- Deliverable code
- Users' installation and operation manuals

Some more debatable items are:

- Requirements documents
- Design
- Test cases and procedures

If software maintenance or acceptance testing is considered part of the real world, these should be considered process entities as long as they are identifiable and are transformed by the process. This, in fact, is one of the key problems in the traditional task-oriented view of the process. The requirements, for example, must evolve throughout the process as the users' needs and the implementation realities are better understood. A similar evolutionary path applies to the design and test materials, which are also needed for subsequent product enhancement and repair.

Another potential entity is a prototype produced to clarify some design question. Assuming that it is produced, tested, and discarded, it is clearly not an entity even though it may contribute importantly to some new design state. Again, this is not a question of what is right or wrong but of what is most helpful in the particular situation. Prototyping would thus show up as an activity, unless, that is, the prototype ended up being delivered as part of the product. The fact that it isn't an entity does not mean that it isn't important. Even if using a prototype were the only way to produce a quality design, it would still not be an entity. It would, however, be an essential part of the design activity. Thus, even though it may be transformed many times, an entity persists throughout the software system's useful lifetime. If the process being modeled only spans development, then that is the useful lifetime. To avoid overlooking key needs or to reduce emphasis on transient development issues, however, it is wise to attempt to consider the full system lifetime, including subsequent enhancement and repair.

4.4. Producing Entity Process Models

The process of producing an entity process model is relatively straightforward:

- Identify the process entities and their states.
- Define the triggers that cause the transitions between these states.
- Complete the process model without resource constraints — an unconstrained process model (UPM).

- Impose the appropriate limitations to produce a final constrained process model (CPM).

While this procedure is conceptually simple, it can become quite complex when applied to real processes.

5. Example EPM

As a means of illustrating the application of entity process modeling, an example is presented that deals with a realistic component of the software development process. For the example to be of manageable size, only a circumscribed portion of the software life cycle is addressed: the coding and unit testing of a single module. Nevertheless, this example does illustrate many of the features of the software process that make it notoriously complex, such as parallelism of tasks, complex decision criteria regarding the next step, iteration loops, and multiple feedback paths. The example will be increasingly refined through three stages.

In this example, we restrict our attention to the activities occurring between the time when (1) detailed design for the module has been developed, and (2) the module has successfully passed unit testing. There are three entities of interest in this example:

1. Module code
2. Unit tests for the module
3. Test execution and analysis results

As explained above, in general, each entity passes through a series of states during its lifetime. An example time line is presented in Figure 1 for module code. This entity does not exist during the early stages of a project, but at some point it begins to be developed and we say it is in a state "undergoing development." When development stops (at least temporarily to allow testing), we say it enters the state "developed." At some point after the code has been developed (and while the entity module code is in the "developed" state), it will be tested. This typically will result in cycling back through additional development work. Finally, the module passes all unit tests and is promoted to the state of "tested and passed."

A few key points of our approach are highlighted by this example. Entities remain in a state for a positive (non-zero) amount of time, while the transitions between states are represented as taking negligible time. Between the time of its creation and termination, an entity must always be in some particular state. Some states are "active", wherein the entity is being created or transformed in some way, such as undergoing development. The remaining states are "passive", wherein the entity remains in the same condition, not undergoing any change. Note that although the module code is tested during its time in the "developed" state, it is not changed. Generally, however, it is relatively simple to identify the various entities of a process and enumerate their salient states.

Our work in software process modeling has primarily utilized a commercially available software

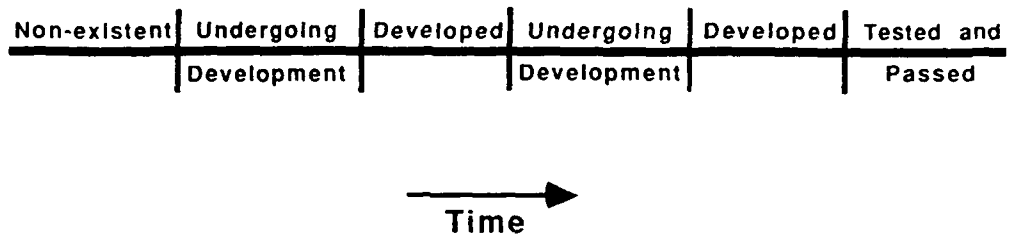

Figure 1: Example Time Line for Module Code Entity

system called STATEMATE.[1] The focus of this paper is on a behavioral modeling perspective, and our approach to behavioral modeling utilizes statecharts [Harel 87, Harel 88a], which are supported by the STATEMATE tool [Harel 88b]. Other aspects of our modeling approach have been described elsewhere [Kellner 88a, Kellner 88b, Kellner 89]. Statecharts are an enhancement to traditional state transition diagrams. The three statecharts presented in this paper were produced on the STATEMATE system.

5.1. Basic Example

Recall that this example considers three entities: module code, module unit tests, and the test execution and analysis results. Figure 2 presents a statechart depicting an entity process modeling view of our example process. The boxes in the diagram represent states, while the lines represent transitions between states. States can have orthogonal components, separated by dashed lines. These orthogonal components represent parallelism (concurrency); for example, the module code, tests, and test execution report all exist concurrently, as illustrated in the upper left quadrant (labeled MODULE_CODE), lower left quadrant (labeled MODULE_TESTS), and upper right quadrant (labeled TEST_EXEC_REPORT) of the diagram, respectively. When the overall process is in a state (such as the large outer state labeled SW_PROCESS), it must also be in a substate in each orthogonal component. We have also included components in the lower right quadrant for upstream entities and downstream entities. These are simply placeholders to illustrate where the rest of the software process would be depicted, and would include states for the entities requirements, designs, system builds, integration tests, etc.

We are using the constructs of an existing tool to depict examples of entity process models. Because STATEMATE does not offer an "entity" construct, we are forced to represent the entities themselves as high-level orthogonal state components, as shown by the major quadrants in the figure. Admittedly, this is a bit awkward, but it only occurs at the top level. At all lower levels, states in the statechart do indeed depict the various states of these entities.

[1]STATEMATE is a trademark of i-Logix, Inc., Burlington, MA.

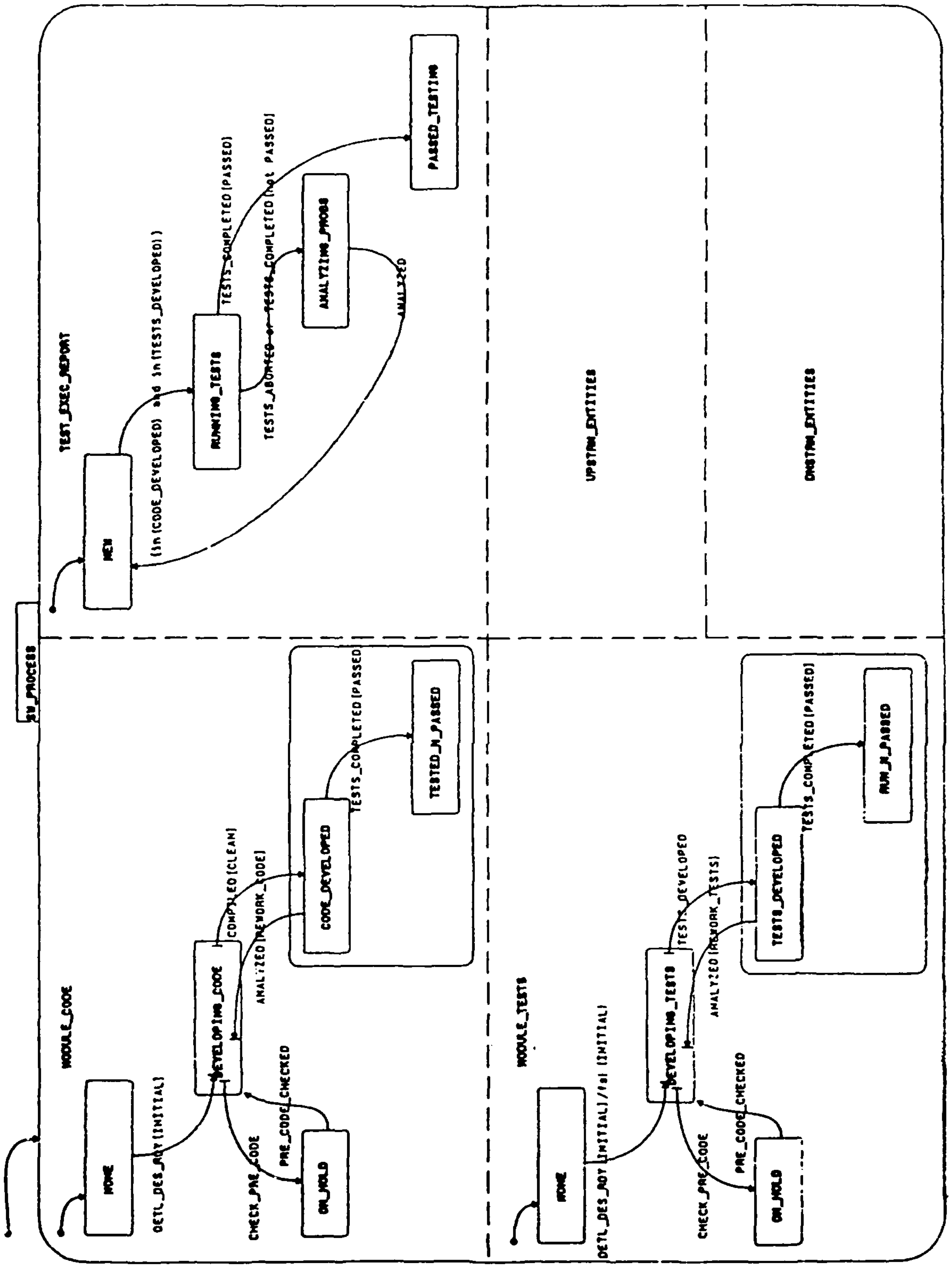

Figure 2: Basic EPM Example

The state transitions are represented by lines whose labels define the trigger for making the transition. Default transitions (beginning with a small dot) lead into the outer state, into the NONE substates of MODULE_CODE and MODULE_TESTS, and into the NEW substate of TEST_EXEC_REPORT. This indicates that the process begins with the entities in those states.

The states within the MODULE_CODE quadrant represent the states of the code entity, the first being "none" or non-existent. When the detailed design is initially ready, the trigger DETL_DES_RDY[INITIAL] is satisfied, and the entity transitions to the DEVELOPING_CODE state. Triggers can be composed of event expressions guarded by condition expressions; conditions evaluate to a Boolean value, and condition expressions are enclosed in brackets such as [INITIAL]. This trigger is interpreted as the event DETL_DES_RDY (detailed design for the module is ready) when INITIAL is true (INITIAL indicates that the coding starts from scratch). A similar explanation applies to the transition from DEVELOPING_CODE to CODE_DEVELOPED: take it at the instant the code is compiled, as long as it was a "clean" compilation. As noted above, testing occurs while code is in the CODE_DEVELOPED state. If analysis of the tests reveals a need to rework the code, we transition back to DEVELOPING_CODE and iterate again. Eventually, the tests will be completed and passed, leading to the final state. The last two states (CODE_DEVELOPED and TESTED_N_PASSED) are enclosed in an unnamed state; this is primarily a matter of style, although it will prove useful later. Finally, this example introduces an ON_HOLD state. Frequently, during development, it is necessary to go back to some earlier entity, such as requirements, to recheck or clarify something, or to consider modifying the design as a result of something uncovered during development. Thus, while in DEVELOPING_CODE, the code entity may transition to the ON_HOLD state for a time, then return to where development left off.

The situation for the entity MODULE_TESTS is almost identical. However, the entity TEST_EXEC_REPORT warrants additional explanation. The various states of this entity result from applying a specific version of the unit tests entity (MODULE_TESTS) to a specific version of the module code entity (MODULE_CODE). Thus, each instance of TEST_EXEC_REPORT really corresponds to a specific code and test pair and must thus be a separate entity in its own right, which represents a combination of the other two entities. Of the four states of this compound entity, two are active: RUNNING_TESTS and ANALYZING_PROBS. Notice the synchronization between the state transitions of the three entities. For example, TEST_EXEC_REPORT can transition from NEW to RUNNING_TESTS only when MODULE_CODE is in its CODE_DEVELOPED state and MODULE_TESTS is simultaneously in its TESTS_DEVELOPED state. Similarly, the trigger TESTS_COMPLETED[PASSED] causes all three entities to progress to their final states simultaneously.

Here the three entities MODULE_CODE, MODULE_TESTS, and TEST_EXEC_REPORT each have five, five, and four states, respectively. Of the resulting 100 potential system states (5 x 5 x 4), only a relatively small number are possible, however, as determined by the various state transition conditions.

5.2. Example with Feedback

The first level of refinement to this model is presented in Figure 3. This version of the example is much more representative of the realities of software processes because it adds many feedback paths to the process. For instance, while analyzing problems uncovered during testing, it may become apparent that revisions are needed to entities developed earlier, such as designs or even requirements. We indicate this with the condition REWORK_PRE_CODE taking the value true. The upper right quadrant of the diagram (TEST_EXEC_REPORT) shows this with a transition out of ANALYZING_PROBS and into a state called HOLD. The transition out of ANALYZING_PROBS is triggered by the event ANALYZED, and leads to a condition connector (circle containing a letter c); multiple branches lead out from there with additional triggers. This further illustrates the representation of decision logic. If the REWORK_PRE_CODE flag is true, the path to HOLD is taken; otherwise, the path directly to NEW is taken. The transition to HOLD illustrates another feature of statecharts as well. The label on the transition ends with /TR!(REWORK_CODE);TR!(REWORK_TESTS) indicating an action to be performed when the transition is taken. This is a simple action of assigning the value "true" to the two rework flags, ensuring that the code and tests will be reexamined after the revisions to earlier work are completed. The transition out of HOLD is taken when DETL_DESIGN_RDY occurs, meaning the detailed design is again ready. This same event triggers transitions in the states of the other two entities, namely the uppermost transitions back into DEVELOPING_CODE and DEVELOPING_TESTS. The INITIAL flag can be set again to indicate a need to throw out the previous work and start fresh because of wholesale changes to the design.

Another example of feedback results when problems arise downstream, such as during integration testing. Analysis of such problems may result in a need to rework the module code or unit tests (and corresponding values of REWORK_CODE and REWORK_TESTS). This sort of feedback is represented by the event LATER_FEEDBACK, and may result in a transition from TESTED_N_PASSED back to DEVELOPING_CODE, and a similar transition for the tests. Other combinations of events and conditions lead to demoting the state of the code from TESTED_N_PASSED back to CODE_DEVELOPED. This happens when the code itself does not need to be reworked, but the tests do — meaning that the new tests will have to be applied to the existing code. Again, a similar situation arises for the tests.

The addition of these various feedback situations certainly complicates the model. However, that is unavoidable since the real software process is vastly complicated by this feedback. Many other software process models systematically avoid illustrating this feedback precisely because it is so complex. However, our approach to modeling allows us to precisely capture and describe this behavior.

A point about this example is worth making at this time. We are not proposing that this example is ***the*** software process. It is a stylized example of ***a*** process which could reasonably be used to develop a module. It has been designed to be representative of actual aspects of software processes, and to illustrate our modeling approach; but it is not dogma. We ask the readers to view it in that light, and to bear with us if they would prefer a slightly different process from that described here.

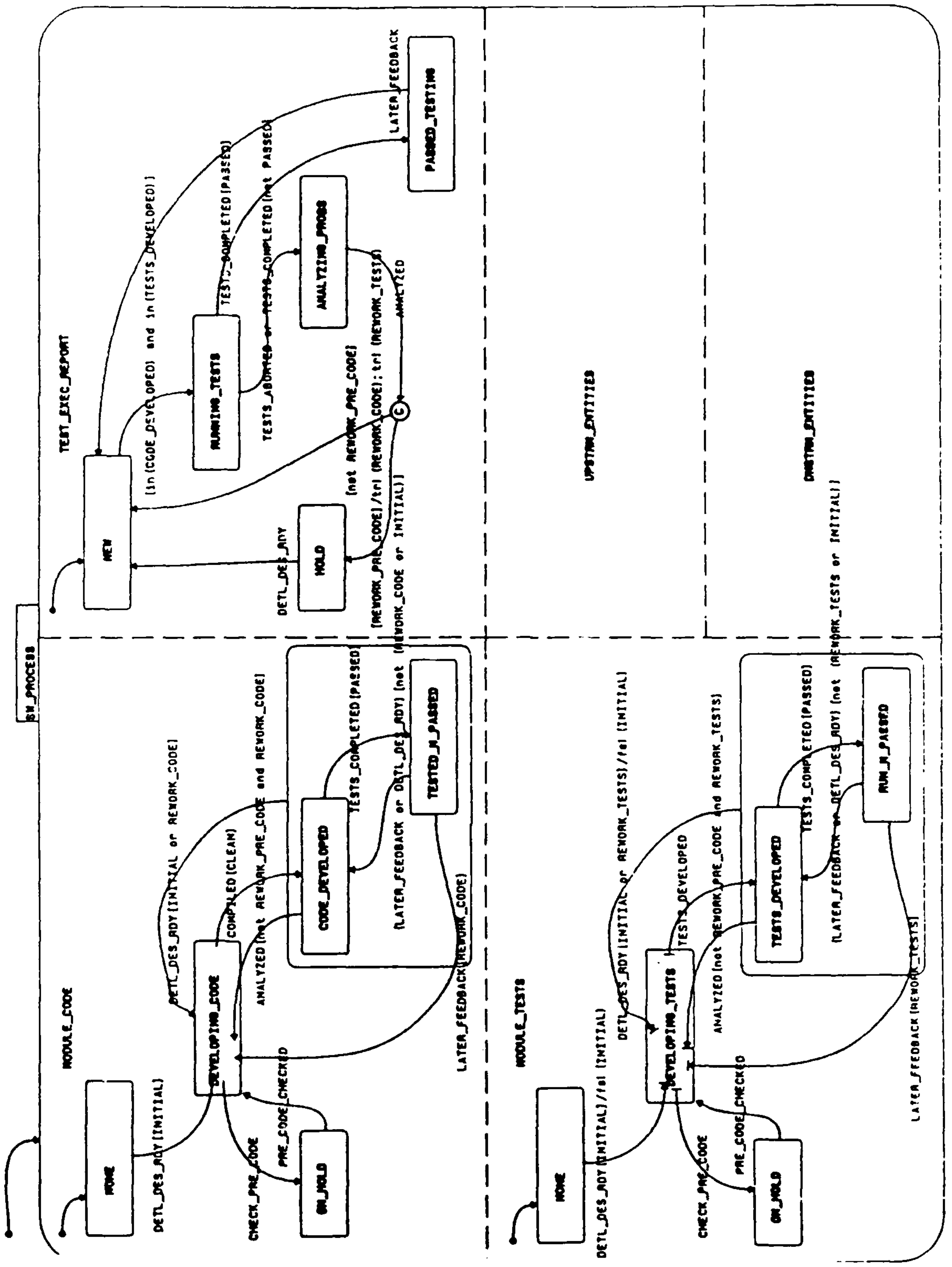
SW_PROCESS
TEST_EXEC_REPORT
NEW
HOLD
RUNNING_TESTS
ANALYZING_PROBS
PASSED_TESTING
C
[in(CODE_DEVELOPED) and in(TESTS_DEVELOPED)]
TESTS_COMPLETED[PASSED]
TESTS_ABORTED or TESTS_COMPLETED[not PASSED]
LATER_FEEDBACK
ANALYZED
[not REWORK_PRE_CODE]
[REWORK_PRE_CODE]/tr! (REWORK_CODE);tr! (REWORK_TESTS)
[REWORK_PRE_CODE]/tr! (REWORK_CODE or INITIAL)]
DETL_DES_RDY
UPSTRM_ENTITIES
DNSTRM_ENTITIES
[not (REWORK_TESTS or INITIAL)]
MODULE_CODE
NONE
DETL_DES_RDY[INITIAL]
DEVELOPING_CODE
CODE_DEVELOPED
TESTED_N_PASSED
ON_HOLD
CHECK_PRE_CODE
PRE_CODE_CHECKED
COMPILED[CLEAN]
ANALYZED[not REWORK_PRE_CODE and REWORK_CODE]
DETL_DES_RDY[INITIAL or REWORK_CODE]
TESTS_COMPLETED[PASSED]
[LATER_FEEDBACK or DETL_DES_RDY][not (REWORK_CODE)]
LATER_FEEDBACK[REWORK_CODE]
MODULE_TESTS
NONE
DETL_DES_RDY[INITIAL]/fs! (INITIAL)
DEVELOPING_TESTS
TESTS_DEVELOPED
RUN_N_PASSED
ON_HOLD
CHECK_PRE_CODE
PRE_CODE_CHECKED
TESTS_DEVELOPED
ANALYZED[not REWORK_PRE_CODE and REWORK_TESTS]
DETL_DES_RDY[INITIAL or REWORK_TESTS]/fs! (INITIAL)
TESTS_COMPLETED[PASSED]
[LATER_FEEDBACK or DETL_DES_RDY][not (REWORK_TESTS)]
LATER_FEEDBACK[REWORK_TESTS]

Figure 3: EPM Example With Feedback

5.3. Further Refinement

The final refinement to this example illustrates how EPMs can be further refined to show additional detail. Figure 4 presents a decomposition of the DEVELOPING_CODE state. For the purposes of this example, we assume that the module is written in a compiled language, and that it is compiled as a unit. Also, for illustrative purposes, we chose to distinguish between initial coding (starting with a clean slate), and revisions to existing code. DEVELOPING_CODE is decomposed into two primary substates: WRITE_CODE and COMPILE_CODE. The former is further decomposed. This lowest level of detail illustrates that INITIAL_CODE is composed of planning (PLAN_CODE) and entering code (ENTER_CODE). These activities can be carried out in an unspecified concurrent fashion, such as interleaving them.

The transition labeled RDY_TO_COMPILE illustrates another aspect of statecharts. This transition leading from WRITE_CODE means that it is a valid transition out from any of the substates of WRITE_CODE. One last statechart feature is the H^* symbol at the lower left corner of the diagram. This is called a deep history connector, and it is connected to the ON_HOLD box, which was reached when code development was interrupted by the need to perform some check at the design or requirements level. The H^* means return to the exact configuration of states within DEVELOPING_CODE that the process was in when the last transition was taken out of DEVELOPING_CODE. This is the destination of the transition PRE_CODE_CHECKED coming back from ON_HOLD; thus the process picks up precisely where it left off when it was interrupted.

As noted above, these statecharts were produced using the STATEMATE system. In addition to offering a convenient representation formalism, the system supports automated analysis of a variety of aspects of completeness, correctness, and consistency. These include tests for reachability, nondeterminism, deadlocks, and race conditions. The process can also be simulated through interactive, animated simulation from the diagrams. Exploration of these issues is beyond the scope of this paper, but is reported elsewhere [Kellner 88b, Kellner 89].

6. Scheduling Considerations

6.1. The Unconstrained Process Model

Entity process models can be used for schedule planning and analysis. The example EPM developed above is called unconstrained because it does not include any consideration of resource constraints in performing tasks and making transitions between states. Thus, it describes the transitions based on their logical interconnections only, without being burdened by issues of resource constraints.

The EPM can be used to derive what we term an unconstrained process model (UPM), which is a schedule for the unconstrained case. Returning to the module development example presented above in Figure 2, Table 1 shows a set of planned times for each task. These tasks correspond to the active states in the statechart model. The basic plan forecasts that after initial development of code and tests, test execution will uncover errors calling for the rework of both code and tests at half their initial effort level. The second round of testing will uncover more errors, but only in the code, requiring one-quarter the initial effort to correct. The tests will then be passed on the

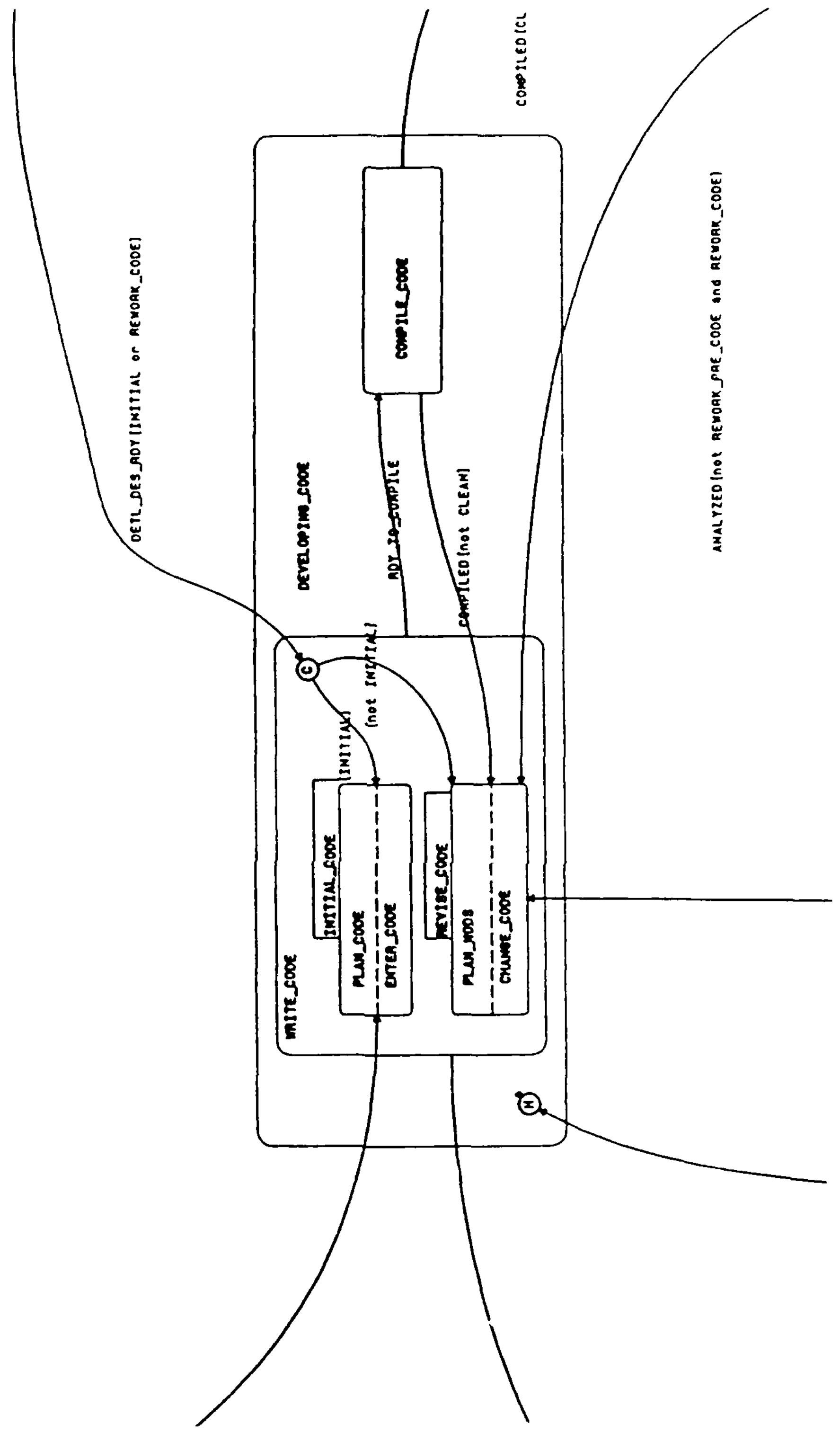

Figure 4: EPM Example — DEVELOPING_CODE Details

third round. We suppose that each of these tasks is a one-person task that cannot be distributed among multiple workers.

Table 1: Resource Requirements for Examples

	Hours
Round 1	
Developing Code	12
Developing Tests	8
Running Tests	1
Analyzing Problems	3
Round 2	
Developing Code	6
Developing Tests	4
Running Tests	1
Analyzing Problems	2
Round 3	
Developing Code	3
Running Tests	1
-- Passed --	

The UPM resulting from these time estimates applied to the EPM of Figure 2 is presented in Figure 5, in the form of a Gantt chart. The chart illustrates when each entity is in each of its four respective states (the ON_HOLD states were not used in this illustration). Those time bars with numbers above them represent the time in the active states, as forecast in the table. Certainly, it is possible to reduce this chart to showing only the active states, but the form shown is more complete. Notice that the tests will be passed in 29 hours, although 41 work hours are required to accomplish the work. The graph at the bottom of the UPM chart shows the resource requirements (in number of personnel) for each time unit.

Since the EPM and UPM are independent of resource constraints, they present a robust representation of task relationships and furnish a useful framework for analyzing process problems and determining where added resources can be most effective.

6.2. The Constrained Process Model

These models also provide the basis for establishing a constrained process model (CPM). Since real software organizations have limited resources, some tasks may have to wait for personnel to become available to accomplish them. The CPM is thus produced by adjusting task timing to obtain the overall results desired, subject to the resource constraints. A typical objective would be to complete the process in the shortest time. The UPM revealed that the process could be

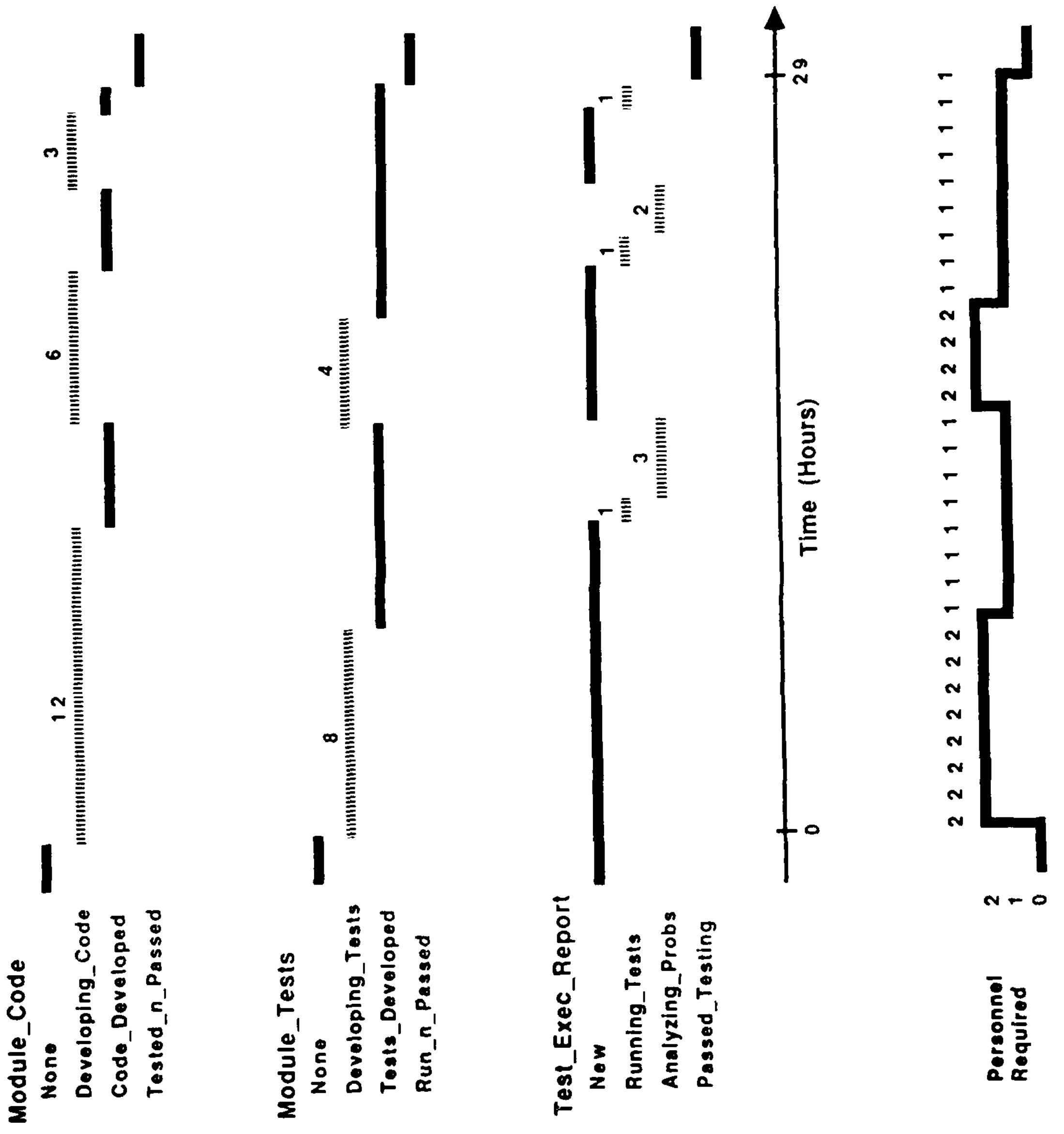

Figure 5: Example Unconstrained Process Model (UPM)

completed in 29 hours, but required two workers during 12 of those hours. Suppose now that only one worker is available. Figure 6 shows one possible schedule for this process, using only one worker. The shortest possible time with this resource constraint is 41 hours, and other schedules can be found that will yield this time. For example, test development could be done before code development.

6.3. Schedule Management

These models can be valuable tools for schedule management. Although the EPM is much more detailed and general than a typical precedence network (such as those used for CPM or PERT), some of the same analyses can be applied. For example, one can see from the UPM (Figure 5) that there is slack time for initial test development: it can begin any time between time 0 and 4 without delaying completion of the process. On the other hand, initial code development is on the critical path; if any way could be found to speed up that task, overall completion would occur sooner. These charts also indicate that the addition of a second worker at certain points could speed up completion.

The models are also useful during process execution. A common software development problem is the occurrence of a crisis delaying completion of a key task. The models allow management to determine if this task is on the critical path. If it is, then the completion schedule is threatened, and management can use the models to assess the available corrective actions. By examining the models, it becomes apparent whether added resources could help and where they should be applied to rebalance the schedule. It should also be noted that although our example utilized a single resource type, this is not a methodological limitation. Multiple resources can be required for each task, including different types of personnel, special talents, or unique equipment. Separate resource requirements charts can be developed for each and considered in developing the CPM.

Project schedules and checkpoints can also be obtained from the CPM for process tracking and control. As can be seen from the example, the checkpoints consist of monitoring the various entity states. Assuming that the entity states are properly defined as discrete measurable conditions of real objects, these checkpoints should be relatively easy to monitor.

This does not mean that examination of work content is not important. On the contrary, it is essential to include quality-directed process tasks in software plans to ensure that the work is properly done. This might require that inspections be held of all important work products and that their completion result in an "inspected" state, which could be monitored by management. Management could thus be thoroughly informed of quality status without having to wade through voluminous reports or specifications.

This does not mean that all management tracking should be restricted to examining entity states. It is highly desirable for managers to be personally familiar with many aspects of the work of their people. This should include review of the technical approach, quality evaluations, quantity measures of the work performed, resource utilization, and productivity. To evaluate schedule performance, however, tracking of entity state progress against plan is precise and efficient.

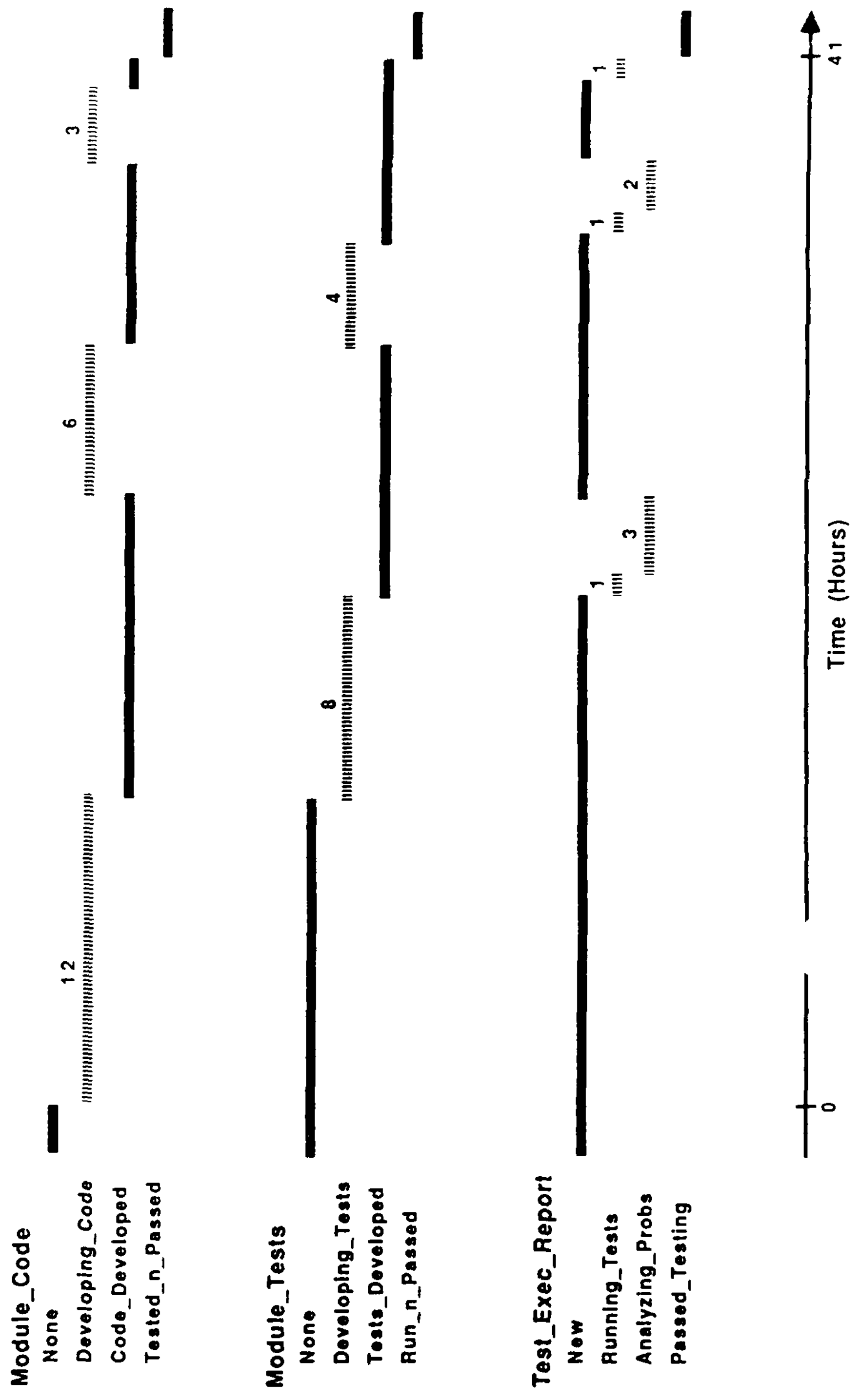

Figure 6: Example Constrained Process Model (CPM)

7. Conclusions

Perhaps the most attractive feature of EPMs is the new forces they generate:

- The prime entities of the software process are seen as persistent objects. With requirements, for example, this makes it clear that the requirements must evolve throughout product development.
- A focus on states facilitates the tracking of 100% completed items rather than vague partial task completions.
- The UPM/CPM duality assists in adjusting for crises without bypassing essential elements of the process.
- The use of the UPM/CPM pair simplifies initial scheduling and planning and permits simple adjustments to conform to new demands and available resources.

This approach appears to solve some of the problems of using task-based process models for processes with highly dynamic task behavior. Since the states of the basic process entities generally behave in a more orderly fashion than the tasks, they can provide a more robust basis for process definition and structure. As a result of these advantages, EPMs should thus help us to better understand software processes and to plan and manage them more effectively.

By using a large-scale process representation that both consistently represents actual process behavior and can be refined to progressively greater levels of detail, better understanding and more accurate control are possible.

In addition, our EPM models focus on the dynamic behavior of a process and its impacts on the relevant entities. Task oriented models do not generally specify the details of dynamic behavior, meaning that they cannot be directly enacted or used for schedule planning. Our EPMs, based on statecharts, are formal and enactable — in that we are able to run interactive, animated simulations of our EPMs with STATEMATE, as well as perform automated tests and analyses. The value of enactable models is addressed more fully elsewhere [Hansen 88].

References

[Basili 75]	Basili, V. R., and A. J. Turner. Iterative Enhancement: A Practical Technique for Software Development. *IEEE Transactions on Software Engineering* SE-1(4), December 1975.
[Boehm 86]	Boehm, Barry W. A Spiral Model of Software Development and Enhancement. *ACM Software Engineering Notes* 11(4): 14-24, August 1986.
[Feiler 86]	Feiler, Peter H., and Gail E. Kaiser. *Granularity Issues in a Knowledge-Based Programming Environment.* Tech. Memo CMU/SEI-86-TM-11, DTIC: ADA182981, Software Engineering Institute; Carnegie Mellon University, September 1986.
[Feiler 88]	Feiler, Peter H., and Roger Smeaton. *Managing Development of Very Large Systems: Implications for Integrated Environment Architectures.* Technical Report CMU/SEI-88-TR-11, ADA197671, Software Engineering Institute; Carnegie Mellon University, May 1988.
[Fourth 88]	Fourth International Software Process Workshop: Representing and Enacting the Software Process. May 11-13, 1988 Held at Moretonhampstead, Devon, UK.
[Hansen 88]	Hansen, Gregory A., and Marc I. Kellner. Software Process Modeling: The Triad Approach. In *Proceedings of the Sixth Symposium on Empirical Foundations of Information and Software Sciences.* October 1988.
[Harel 87]	Harel, David. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8(3): 231-274, June 1987.
[Harel 88a]	Harel, David. On Visual Formalisms. *Communications of the ACM* 31(5): 514-530, May 1988.
[Harel 88b]	Harel, David, et al. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. In *Proceedings of the 10th International Conference on Software Engineering,* pages 396-406. IEEE, 1988.
[Humphrey 88]	Humphrey, Watts S. The Software Engineering Process: Definition and Scope. In *Proceedings of the 4th International Software Process Workshop: Representing and Enacting the Software Process,* pages 34-35. ACM, 1988.
[Jackson 83]	Jackson, Michael A. *System Development.* Prentice-Hall International, Englewood Cliffs, NJ, 1983.

[Kellner 88a] Kellner, Marc I.
Representation Formalisms for Software Process Modeling.
In *Proceedings of the 4th International Software Process Workshop: Representing and Enacting the Software Process*, pages 43-46. ACM, 1988.

[Kellner 88b] Kellner, Marc I., and Gregory A. Hansen.
Software Process Modeling.
Technical Report CMU/SEI-88-TR-9, DTIC: ADA187137, Software Engineering Institute; Carnegie Mellon University, May 1988.

[Kellner 89] Kellner, Marc I., and Gregory A. Hansen.
Software Process Modeling: A Case Study.
In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, Vol. II — Software Track*, pages 175-188. IEEE, 1989.

[Lehman 87] Lehman, M. M.
Process Models, Process Programs, Programming Support.
In *Proceedings of the 9th International Conference on Software Engineering*, pages 14-16. IEEE, 1987.

[Osterweil 87] Osterweil, Leon.
Software Processes Are Software Too.
In *Proceedings of the 9th International Conference on Software Engineering*, pages 2-12. IEEE, 1987.

[Phillips 88] Phillips, Richard W.
State Change Architecture: A Protocol for Executable Process Models.
In *Proceedings of the 4th International Software Process Workshop: Representing and Enacting the Software Process*, pages 74-76. ACM, 1988.

[Phillips 89] Phillips, Richard W.
State Change Architecture: A Protocol for Executable Process Models.
In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, Vol. II — Software Track*, pages 154-164. IEEE, 1989.

[Rombach 89] Rombach, H. Dieter, and Leo Mark.
Software Process & Product Specifications: A Basis for Generating Customized Software Engineering Information Bases.
In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, Vol. II — Software Track*, pages 165-174. IEEE, 1989.

[Royce 70] Royce, Winston W.
Managing the Development of Large Software Systems: Concepts and Techniques.
In *Proceedings of IEEE WESCON*, pages 1-9. IEEE, August 1970.

[Royce 87] Royce, Winston W.
Managing the Development of Large Software Systems.
In *Proceedings of the 9th International Conference on Software Engineering*, pages 328-338. IEEE, 1987.

[Williams 88a] Williams, Lloyd G.
Software Process Modeling: A Behavioral Approach.
In *Proceedings of the 10th International Conference on Software Engineering*, pages 174-186. IEEE, 1988.

[Williams 88b] Williams, Lloyd G.
A Behavioral Approach to Software Process Modeling.
In *Proceedings of the 4th International Software Process Workshop: Representing and Enacting the Software Process*, pages 108-111. ACM, 1988.

UNLIMITED, UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-89-TR-2	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-89-TR-2
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INSTITUTE 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) SEI JPO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY SOFTWARE ENGINEERING INSTITUTE JPO PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
	N/A	N/A	N/A

11. TITLE (Include Security Classification)
SOFTWARE PROCESS MODELING: PRINCIPLES OF ENTITY PROCESS MODELS

12. PERSONAL AUTHOR(S)
HUMPHREY, KELLNER

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
FINAL	FROM ______ TO ______		

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A DEFINED SOFTWARE PROCESS IS NEEDED TO PROVIDE ORGANIZATIONS WITH A CONSISTENT FRAMEWORK FOR PERFORMING THEIR WORK AND IMPROVING THE WAY THEY DO IT. AN OVERALL FRAMEWORK FOR MODELING SIMPLIFIES THE TASK OF PRODUCING PROCESS MODELS, PERMITS THEM TO BE TAILORED TO INDIVIDUAL NEEDS, AND FACILITATES PROCESS EVOLUTION. THIS PAPER OUTLINES THE PRICINCIPLES OF ENTITY PROCESS MODELS AND SUGGESTS WAYS IN WHICH THEY CAN HELP TO ADDRESS SOME OF THE PROBLEMS WITH MORE CONVENTIONAL APPROACHES TO MODELING SOFTWARE PROCESSES.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] SAME AS RPT ☐ [✓ DTIC USERS]	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) (412) 268-7630 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

UNLIMITED, UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE